LEHRSTUHL FÜR RECHNERARCHITEKTUR UND PARALLELE SYSTEME

Praktikum Rechnerarchitektur

Gruppe 110 – Abgabe zu Aufgabe A315

Sommersemester 2020

Maksym Bondarenko Ihor Kudryk Aiina Tikhonova

1 Einleitung

Die Aufgabe unserer Gruppe besteht darin, die folgende Funktion mit x86_64 Assembler zu implementieren:

$$f(x, y) = x^y$$

Dabei ist es für rationale $y, x > 0$ anzunehmen. Für ganze y gilt keine Einschränkung für x.

Der Schlüsselfaktor ist, dass die Variablen auch rationale Werte annehmen können. Aus diesem Grund eignet sich die Implementierung nur mit Hilfe von y-fachen Multiplikation für diese Aufgabe nicht.

Außerdem sind in der Impelemtierung Befehle für Exponential- und Logarithmusberechnungen untersagt Stattdessen müssen diese Befehle wenn nötig selbst implementiert werden. Spezielle Werte wie Plus- oder Minusunendlichkeit als EIngaben sind theoretisch auch möglich.

Diese Aufgabe ist von besonderem Interesse, da genau die Berechnungen mit Gleitkommazahlen bereits in den siebziger Jahren des letzten Jahrhunderts und die Konstruktion von Zahlen in einem rationalen Grad eine mühsame Aufgabe war.

Im Weiteren wird gezeigt, was sich hinter der Funktion $pow(x, y)$ [1] [2] in Standardbibliotheken mehrerer Programmiersprachen steckt. Für auf dem ersten Blick einfache Berechnung wird eine Approximation durch die Taylorreihe benötigt, sowie weitere Ansätze. Bei geschickter Implementierung kann die Methode präzise Genauigkeit und Geschwindigkeit liefern.

2 Lösungsansatz

Idee zur Lösung des Problems x^y besteht darin, es in folgende Teilaufgabe zu unterteilen:

1. Da es nicht trivial ist, mit einer rationalen Zahl y die Funktion $f = x^y$ zu implementieren, bietet sich die nächste Idee an:

$$\boldsymbol{f(x,y) = x^y = e^{y \cdot lnx}} \tag{1}$$

   Folgende Ansätze werden für die Entwicklung der Formel erforderlich:

   - Taylor-Entwicklung [4] für e^z:

$$\begin{aligned} e^z &= 1 + \frac{z}{1!} + \frac{z^2}{2!} + \frac{z^3}{3!} + ... \\ &= 1 + \frac{z}{1} \cdot (1 + \frac{z}{2} \cdot (1 + \frac{z}{3} \cdot (...))) \end{aligned} \tag{2}$$

   - Logarithmus-Regel [6]:

$$a^b = e^{b \cdot ln(a)} \tag{3}$$

   Die einzige Funktion, die nach der Ableitung gleich bleibt, ist die Exponentialfunktion. Mit dieser Idee kann jegliche Zahl mit Hilfe von der Taylorreihenentwicklung potenziert werden.

2. Laut der Aufgabenstellung sind nur primitive Operationen wie Multiplikation, Addition usw. erlaubt. Das führt zu einer weiteren Approximation [7] vom natürlichen Logarithmus. n ist eine konstante Größe. Je größer diese Konstante ist, desto genauer ist das Ergebnis und desto mehr Schritte sind benötigt, was andererseits zum Verlust bei der Ausführungsgeschwindigkeit führt.

$$ln(x) = 2 \sum_{k=1}^{n} \frac{((x-1)/(x+1))^{2k-1}}{2k-1} \tag{4}$$

   Der Wert des Logarithmus wird mit Hilfe einer sog. Erweiterung der Logarithmusfunktion [4] berechnet, die ebenfalls aus der Taylorentwicklung hergeleitet wurde.

3. Im Weiteren muss die Potenz der Eulerschen Zahl berechnet werden: multiplizieren y mit Ergebnis aus dem Schritt 2 : $y \cdot ln(x)$

4. Im letzten Schritt wird die Eulersche Zahl auf den im vorherigen Schritt 3 berechneten Wert potenziert: e^a, wo $a = y \cdot ln(x)$

Optimierungsansatz:

1. Sodass das Programm wenig Zeit mit Berechnung von trivialen Fällen verbringt, werden sie am Anfang betrachtet. Ohne Durchführung der aufwändigen Berechnungen, kann der entsprechende Eingabeparamtere in xmm-Register geladen und da näher untersucht werden. Obwohl die allgemeine Perfomaz dabei sinkt, werden in diesem Schritt auch alle Spezialfälle behandelt (z.B. *NaN*, $\pm$ *INFINITY* usw.). Falls die entsprechenden Bedingungen für Randfälle getroffen werden, liefert der Algorithmus das richtige Ergebnis in einer kurzer Zeit.

2. Nach einer gründlichen Filterung von trivialen Fällen wird der folgende Ansatz verwendet (g bzw. r entsprechen den Teilen vor bzw. nach dem Komma):

$$x^y = x^{g,r} = x^g \cdot x^{0,r} \tag{5}$$

3. Nach einer Konvertierung Scalar Double-Precision Floating Point $\rightarrow$ Doubleword Integer mit Hilfe von einem $cvttsd2si$-Befehl [3] wird g von dem Teil nach dem Komma abgetrennt. r wird als eine Differenz der angegebenen Basiszahl mit dem entsprechenden abgerundeten Wert berechnet. Der Rest muss folgendermaßen begrenzt sein: $0 \leq r < 1$.

4. Für das ganze Teil der Potenzzahl g wird ein naiver Ansatz verwendet: es wird zu einem separaten Programm *pow_int* gesprungen, wo x in einer Schleife mit sich selbst multipliziert wird. Es macht keinen Unterschied, ob die gegebene Basis eine Gleitkommazahl oder ganzzahlig ist.
   Falls g negativ ist, wird folgende Kalkulation durchgeführt:

$$a^{-1} = \frac{1}{a} \tag{6}$$

   D.h. das Ergebnis von der Schleife wird invertiert und damit sind die Berechnungen von *pow_int* fertig.

5. In diesem Schritt wird festgestellt, ob der Rest r gleich 0 ist, d.h. ob die angegebene Potenzzahl y ganzzahlig ist. In diesem Fall kann es zu dem Ende des Programms f gesprungen werden, da keine weiteren Berechnungen mehr nötig sind.

6. In dem Gegenfall $r > 0$ wird im Weiteren der Rest r als die Eingabe x angenommen. Wegen des kleinen Wertes von r werden sowohl Genauigkeit, als auch Geschwindigkeit deutlich besser.

7. Nach der Ausführung von dem Hauptansatz (1) nach der Taylorreihenentwicklung wird dessen Ergebnis mit dem Ergebniswert von *pow_int* multipliziert (5).

Damit sind alle Fälle berücksichtigt und die Berechnungen sind fertig.

Alternativer Lösungsansatz:
Eine weitere Alternative zur Berechnung von Logarithmus wäre eine Approximation durch die weitere Simpsonregel [9] für $n = 10$ äquidistante Stützpunkte u_i, $i \in [0..9]$:

$$ln(x) = \int_1^x y(u)du \approx \frac{h}{3}[y(u_0) + y(u_{10}) + 2(y(u_2) + y(u_4) + y(u_6) + y(u_8)) \\ +4(y(u_1) + y(u_3) + y(u_5) + y(u_7) + y(u_9))] \tag{7}$$

wobei

$$h = \frac{b-a}{2n} \tag{8}$$

und

$$y(u) = \frac{1}{u} \tag{9}$$

Die Simpsonregel(7), eine der Quadraturmethoden, erlaubt die Ableitungsfunktion von $ln(x)$ näherungsweise zu integrieren. Ergebnis ist eine Approximation von $ln(x)$.

Die entsprechende Implementierung ist im Verzeichnis „Alternativer_Ansatz“ in Dateien „f_alternativ.s“, „ln_alternativ.s“ und „simpson.s“. Allerdings wurde trotz guter Geschwindigkeit im Prozesslauf auf den Simpsonansatz verzichtet. Die Gründe dafür waren Ungenauigkeit für manche Eingaben, unnötige Komplikation und keine Möglichkeit, den Wert $ln(x)$ für Eingaben zwischen 0 und 1 auszurechnen.

3 Genauigkeit

Sodass die Genauigkeit von dem Algorithmus verständlicher gezeigt werden kann, ist es notwendig Vergleichstests durchzuführen. Dafür wurde der Algorithmus mit der Funktion $pow(x, y)$ aus der Standardbibliothek verglichen.

In Tests wurden die Ergebnisse mit 15 Stellen nach dem Komma betrachtet. Die Tabellendaten sind ebenfalls von diesem Format unterstützt.

In den folgenden Tabellen(2, 3) wird die prozentuale Genauigkeit des Algorithmus im Vergleich zu der Standardfunktion $pow(x, y)$ dargestellt. Dabei sind extra ganz unterschiedliche Werte von x und y verwendet, damit die meisten Eingabetypen bedeckt werden können.

Genauigkeit gleich für 15 Gleitkommazahlen
Kleine Differenz bis zu 1%
Große Differenz, ab 1%
Keine Notwendigkeit zur Überprüfung, entsprechend den Projektbedingungen
Die Angaben erfolgen in Prozenten der Abweichung von Standard-C-Funktion

Abbildung 1: Legende für Tabellen

Genauigkeitsvergleich mit Schritt = 100									
X\Y (%)	**2,4253**	**3**	**-5**	**420**	**0,67**	**-0,81**	**347,63**	**-382,74**	**73,31339**
0	0	0	0	0	0	0	0	0	0
-1		0	0	0					
1	0	0	0	0	0	0	0	0	0
3,4	0	0	0	1,39E-13	0	0	1,71E-13	0	4,92E-14
-364,5		0	0	0					
1,24	1,32E-14	0	0	5,78E-13	0	0	3,37E-13	0	3,96E-14
-149		0	0	0					
-4		0	0	0					
1200	29,71%	0	0	0	-42,61%	95,70%	0	0	22,88%
4242	54,98%	0	0	0	-71,55%	357,10%	0	0	44,46%

Abbildung 2: Genauigkeitsvergleich mit Schritt = 100

In der obigen Tabelle 2 wird die Differenz in % für eine Ausführung mit jeweils 100 Schritte in der Logarithmus- und Exponentialfunktion dargestellt.
Mit einer Erhöhung der Schrittanzahl bis 10.000 bei Logarithmusfunktion lassen sich die Werte verbessern (s. Tabelle 3). Dementsprechend wird sich die Geschwindigkeit der Berechnung verschlechtern.

Genauigkeitsvergleich mit Schritt = 10.000									
X\Y (%)	**2,4253**	**3**	**-5**	**420**	**0,67**	**-0,81**	**347,63**	**-382,74**	**73,31339**
0	0	0	0	0	0	0	0	0	0
-1		0	0	0					
1	0	0	0	0	0	0	0	0	0
3,4	0	0	0	0	0	0	1,71E-13	0	4,92E-14
-364,5		0	0	0					
1,24	1,32E-14	0	0	5,78E-13	0	0	3,37E-13	0	3,96E-14
-149		0	0	0					
-4		0	0	0					
1200	1,12E-12	0	0	0	0	0	0	0	8,39E-13
4242	3,30E-04	0	0	0	-0,50%	-0,60%	0	0	0,02%

Abbildung 3: Genauigkeitsvergleich mit Schritt = 10.000

Aus der Tabelle 4 ist es zu sehen, dass auch die Randwerte wie $\pm$ 1, 0, *NaN* und $\pm$ *INFINITY* von der angesetzen Funktion angenommen werden. Die übereinstimmenden bzw. ungenauen Werte mit *Grün* bzw. *Rot* markiert. In der Tabelle liegen die Berechnungsergebnisse für nicht triviale Fälle vor.

Randfälle							
x\y	**nan**	**inf**	**-inf**	**0**	**4,2**	**0,42**	**-0,42**
nan	nan	nan	nan	1	nan	nan	nan
inf	nan	inf	0	1	inf	inf	0
-inf	-nan	inf	0	1			
-1	-nan	1	1	1			
0	nan	0	inf	1	0	0	inf
0,42	nan	0	inf	1			
-0,42	-nan	0	inf	1			
1235346	nan	inf	0	1			
-238149	-nan	inf	0	1			

Abbildung 4: Verhalten mit Randwerten

4 Performanzanalyse

Für die Performanzanalyse wurde die implementierte Funktion mit einem analogen Programm in der Programmiersprache C, der Funktion $pow(x, y)$ aus der Standardbibliothek und der Lösungsalternative (7) verglichen.

Getestet wurde auf einem System mit einem *Intel®Core ™i3-7100U CPU, 2.40GHz x 4, 3,7 GiB Arbeitsspeicher, Ubuntu 18.04.4 LTS, 64 Bit, Linux-Kernel 4.15.0-111-generic.* Kompiliert wurde mit *GCC 7.5.0* mit der Option *-O3.*

In der X-Achse der Grafik (5) stehen Werte für y und in der Y-Achsen die Ausführungszeit in Mikrosekunden. Der Wert x = 3,197170 ist dabei fest für alle Berechnungen. Der Abstand zwischen Messungen beträgt jeweils mehr als eine Sekunde (sleep(1) wird aufgerufen) und alle alle Aufrufe erfolgen mindestens 15 Mal und bilden letztlich einen Durchschnittswert. Dabei handelt es sich um denselben x-Wert und verschiedene y-Werte.

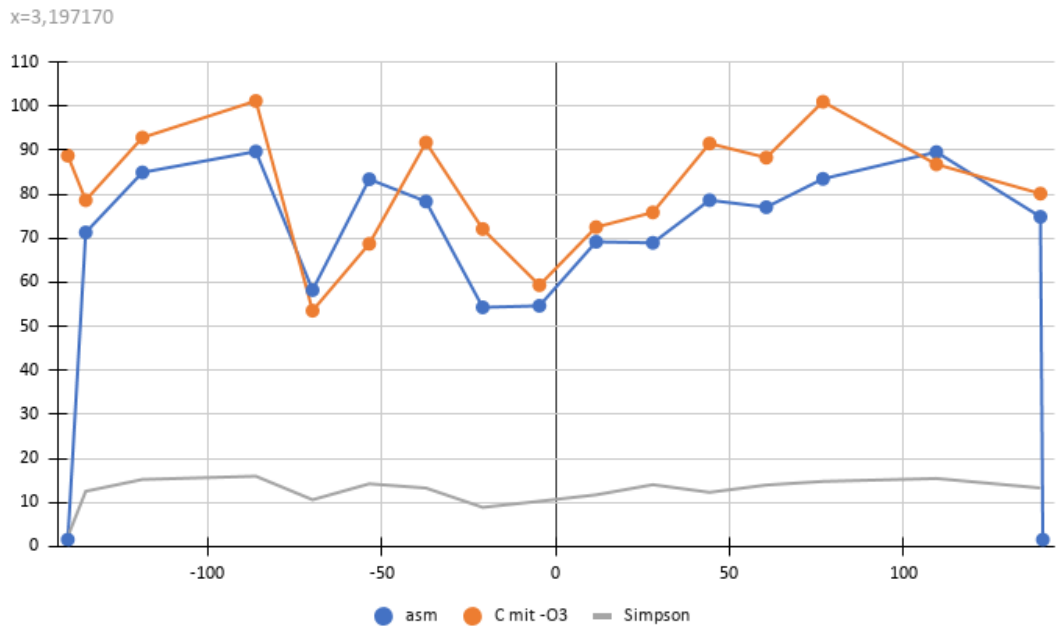

Abbildung 5: Ausführungszeit

Es ist zu bemerken, dass an einigen Stellen die Laufzeit sehr gering ist und zu 0 tendiert . Diese Leistung wird teilweise auch dadurch erreicht, dass die ganzen Zahlen besonders in Betracht gezogen werden(2). Trotz schneller Geschwindigkeit wird der Simpsonansatz nicht verwendet, da die Genauigkeit weniger präzise ist. Sie zu erhöhen ist sehr aufwändig und wird die Situation am Schluss nicht verbessern.

Aus der Abbilding (6) kann geschlossen werden, dass das Verhalten von dem *-O3* C-Programm und dem Assemblerprogramm ähnlich für verschiedene Werte ist. Allerdings ist die Laufzeit der $f(x, y)$-Funktion offensichtlich schneller.

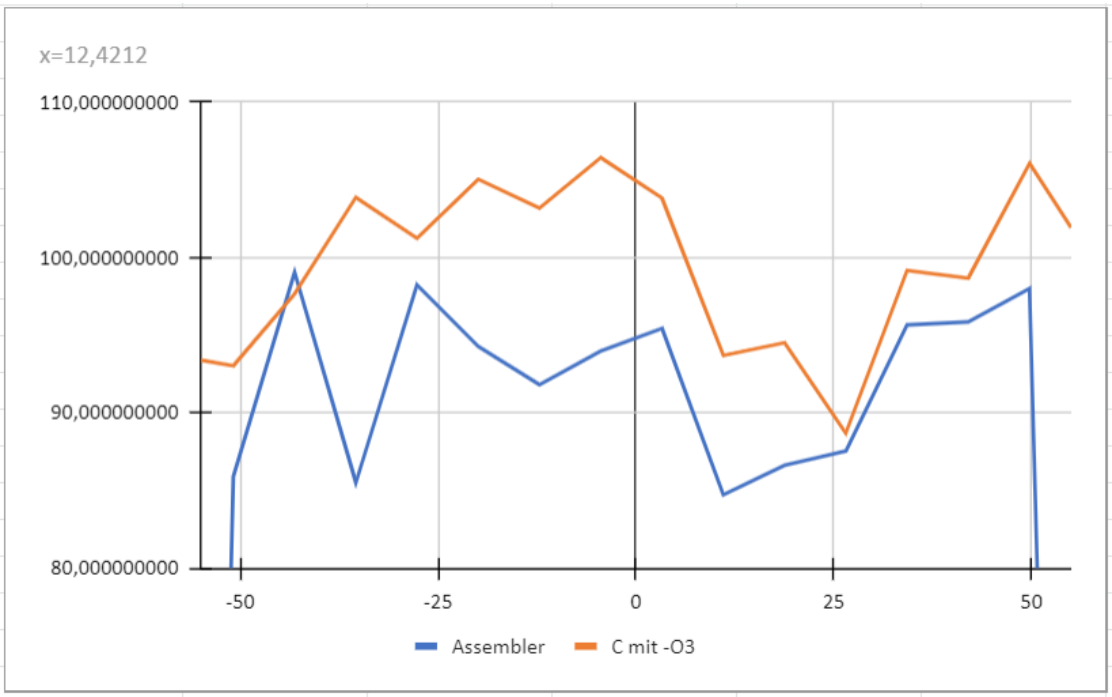

Abbildung 6: Y-Ausführungszeit; X-Werte für y

Da die Methode auf der Taylorreihe-Entwicklung basiert ist, hängen Genauigkeit und Performanz stark von der Anzahl an Schritten n ab. Es kann mithilfe des *perf*-Tools [8] nachvollziehbar gezeigt werden. Hier ist es zu sehen, dass zwischen allen Teilen des Assemblercodes ist das mit Taylorformel implementierte Program den größten Zeitaufwand kostet. Es bedeutet, dass die Logartihmuserweiterung ein sog. Flaschenhals des ganzen Codes ist und deswegen hat sie die größte Auswirkung auf die allgemeine Performanz.

```
Samples: 929  of event 'cycles:ppp', Event count (approx.): 1858000
Overhead  Command  Shared Object      Symbol
   7,64%  main     [kernel.kallsyms]  [k] filemap_map_pages
   6,78%  main     [kernel.kallsyms]  [k] unmap_page_range
   4,31%  main     libc-2.27.so       [.] _dl_addr
   3,23%  main     [kernel.kallsyms]  [k] prepare_exit_to_usermode
   2,91%  main     ld-2.27.so         [.] do_lookup_x
   2,48%  main     ld-2.27.so         [.] _dl_relocate_object
   2,48%  main     main               [.] taylor
   1,94%  main     [kernel.kallsyms]  [k] vma_interval_tree_insert
   1,83%  main     [kernel.kallsyms]  [k] do_syscall_64
```

Abbildung 7: Zeitaufwand für einzelne Teile des Codes

Die folgende Tabelle 8 zeigt den exponentiellen Wachstum der Ausführungszeit mit festen x und y in Abhängigkeit von der konstanten Zahl n aus der Logarithmuserweiterung.

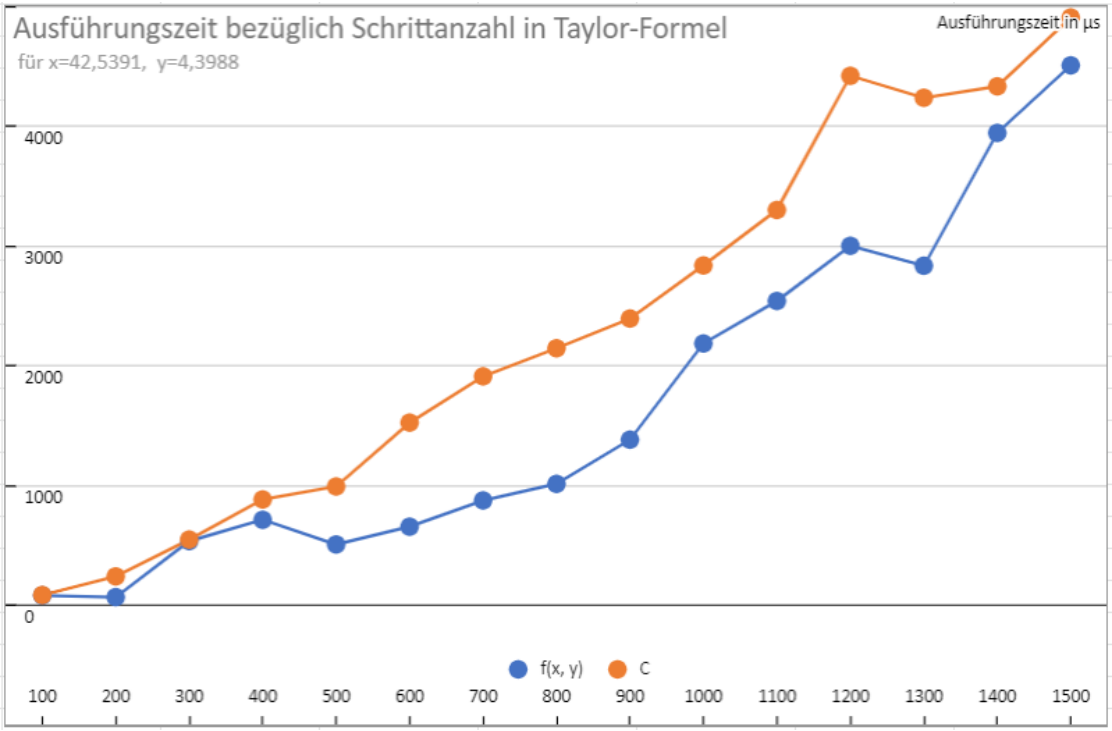

Abbildung 8: Exponentiale Ausführungszeit bezüglich Logarithmuserweiterungsschritt

5 Zusammenfassung und Ausblick

Die Potenzberechnung mithilfe der Taylorreihe-Entwicklung ist wünschenswert, wenn eine schnelle Implementierung der Potenzfunktion für keine großen Double-Werte erforderlich ist. Außerdem gibt es mit einer weiteren Optimisierung ein Potenzial, die Ergebnisse mit keiner Abweichung von der Standardfunktion $pow(x, y)$ in einer kürzeren Zeit zu liefern.

1. Eine der möglichen Verbesserungen des Algorithmus ist die Binäre Exponentiation [5]. Dabei wird das Potenzieren einer reellen Zahl mit einer ganzen Zahl n im Durchschnitt für $O(log(n))$ statt für $O(n)$ Schritte erfolgen.
   In dem Ansatz werden die benötigten Potenzen von x direkt aufmultipliziert. Er wird folgendermaßen in Pseudocode geschrieben werden:

```
// Berechnet x^k
// res … Resultat der Berechnung

function res = bin_exp(x,k)
  res = 1
  while k > 0
    if k mod 2 == 1
      res = res * x
    end-if
    x = x^2
    k = k DIV 2 //Ganzzahlige Division (das Ergebnis wird abgerundet)
  end-while
  return res
end-function
```

Abbildung 9: Quelle: Wikipedia [5]

2. Wie oben erwähnt wurde der Simpsonansatz als eine alternative Lösung betrachtet. In der Abbilding (4) wird gezeigt, bei welchen Fällen die Potenzberechnung relativ fehlerhaft angezeigt wird, nämlich bei großen Eingaben für die Basis x und nicht trivialen y-Eingaben. Wenn die angesetze Taylorformel-Berechnung sich auch mit einer weiteren Schrittsvergrößerung nicht mehr sinvoll verbessern kann, eignet sich für diese Situation mehr die Simpsonregel. Mit mehreren Stützpunkten (mit 1000000 Stützpunkten getestet) wird das Ergebnis genauer als bei Taylorformel mit denselben Parametern. Der Nachteil ist, dass Simpson offensichtlich viel langsamer ist. Deswegen eignet sich für die Implementierung der Potenzfunktion die Taylorreihenentwicklung mit einer Logarithmuserweiterung viel besser.

Literatur

[1] devdocs.io. *pow, powf, powl*. devdocs.io, July 2020. https://devdocs.io/c/numeric/math/pow, visited 2020-07-23.

[2] gnu.org. *19.4 Exponentiation and Logarithms*. gnu.org, July 2020. https://www.gnu.org/software/libc/manual/html_node/Exponents-and-Logarithms.html#Exponents-and-Logarithms, visited 2020-07-23.

[3] Intel Corporation. *Intel 64 and IA-32 Architectures Software Developer Manuals*. Intel Corporation, May 2020. https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html, visited 2020-07-23.

[4] Wikipedia. *Taylor-Formel*. Wikipedia, November 2019. https://de.wikipedia.org/wiki/Taylor-Formel, visited 2020-07-23.

[5] Wikipedia. *Binäre Exponentiation*. Wikipedia, May 2020. https://de.wikipedia.org/wiki/Bin%C3%A4re_Exponentiation, visited 2020-07-23.

[6] Wikipedia. *Logarithmus als Umkehrfunktion der Exponentialfunktion*. Wikipedia, May 2020. https://de.wikipedia.org/wiki/Logarithmus#Als_Umkehrfunktion_der_Exponentialfunktion, visited 2020-07-23.

[7] Wikipedia. *Natural logarithm*. Wikipedia, June 2020. https://en.wikipedia.org/wiki/Natural_logarithm, visited 2020-07-23.

[8] Wikipedia. *Perf Tool*. Wikipedia, June 2020. https://perf.wiki.kernel.org/index.php/Main_Page, visited 2020-07-23.

[9] Wikipedia. *Simpsonregel*. Wikipedia, February 2020. https://de.wikipedia.org/wiki/Simpsonregel, visited 2020-07-23.